Май 2006

Оригинальный материал:
http://www.computer.org/portal/site/computer/menuitem.5d61c1d591162e4b0ef1bd108bcd45f3/index.jsp?&pName=computer_level1_article&TheCat=1005&path=computer/homepage/0506&file=cover1.xml&xsl=article.xsl&

Перевод: Павел Макаров (pavel.makarov@mail.ru)

Можем ли мы делать операционные системы надёжными и безопасными?

Эндрю С. Таненбаум (Andrew S. Tanenbaum), Йоррит Н. Эрдер (Jorrit N. Herder) и Герберт Бос (Herbert Bos),
Университет Врийе, Амстердам

Микроядра – длительное время отвергаемые из-за их меньшей производительности по сравнению с монолитными ядрами – вновь могут вернуться благодаря их потенциально более высокой надёжности, которую многие исследователи сегодня рассматривают как более важную характеристику, чем быстродействие.

Когда в последний раз ваш телевизор ломался или просил вас загрузить из сети немного критических обновлений для программного обеспечения? В конце концов (если он не очень древний), он ведь является просто компьютером с процессором, большим монитором, некоторым количеством аналоговой электроники для декодирования радиосигналов, с парой специфических устройств ввода-вывода – пультом дистанционного управления, встроенным видеомагнитофоном или DVD-приводом – и с целиком загруженным в ПЗУ программным обеспечением.

Этот риторический, в общем-то, вопрос плавно подводит нас к маленькой, но неприятной загадке, которую мы в компьютерной промышленности не любим обсуждать: почему телевизоры, DVD-магнитофоны, MP3 плееры, сотовые телефоны и остальные напичканные программным обеспечением электронные устройства надёжны и безопасны, а компьютеры – нет? Конечно, существует много «причин»: компьютерное «железо» легко поддаётся модернизации, пользователи могут заменять программное обеспечение, индустрия информационных технологий (IT) является слаборазвитой и так далее – но чем дальше мы продвигаемся в эпоху, когда подавляющее большинство компьютерных пользователей не являются «технарями», тем больше и больше это кажется для них неубедительными отговорками.

Потребители ожидают от компьютера того же, что и от телевизора: вы покупаете его, включаете и он прекрасно работает ближайшие 10 лет. Как профессионалы в области IT, мы обязаны принять этот вызов и делать компьютеры столь же надёжными и безопасными, как телевизоры.

Самым страшным «преступником» с точки зрения надёжности и безопасности является операционная система. Хотя прикладные программы и содержат множество дефектов, однако ошибки в них могли бы вызывать только ограниченный вред, если бы их (ошибки) не содержала бы сама операционная система. Вот почему в этой статье мы сконцентрируемся именно на операционных системах.

Однако, прежде, чем перейти к деталям, уместно сказать несколько слов о взаимосвязи между надёжностью и безопасностью. Проблемы с каждой из них часто имеют одну и ту же основную причину: ошибки в программном обеспечении. Ошибка переполнения буфера может вызвать полный отказ системы(это - проблема надёжности), но она может также позволить умело написанному вирусу или червю перехватить управление компьютером (а это уже - проблема безопасности). Хотя мы концентрируемся в первую очередь на надёжности, улучшая её, можно также улучшить и безопасность.

ПОЧЕМУ СИСТЕМЫ НЕНАДЁЖНЫ?

Современные операционные системы имеют две особенности, которые делают их ненадёжными и небезопасными: они огромные и имеют очень слабую локализацию ошибок. Ядро Linux имеет более 2,5 миллионов строк кода, ядро же Windows XP как минимум вдвое больше.

Одно из исследований надёжности программного обеспечения показало, что код содержит от 6 до 16 ошибок на 1000 строк исполняемого кода[1], в то время, как другое исследование даёт плотность от 2 до 75 ошибок на 1000 строк исполняемого кода[2] в зависимости от размера модуля. Используя умеренную оценку в 6 ошибок на 1000 строк исполняемого кода, получаем, что ядро Linux, по всей видимости, содержит около 15'000 ошибок; Windows XP же имеет их как минимум вдвое больше.

Ещё хуже то, что обычно около 70% операционной системы состоит из драйверов устройств, которые имеют долю ошибок в 3÷7 раз выше, чем простой код[3], поэтому упомянутое выше количество ошибок, вероятно, сильно недооценено. Очевидно, что нахождение и исправление всех этих ошибок просто неосуществимо; более того, исправление старых ошибок часто приводит к появлению новых.

Большой размер современных операционных систем означает, что нет ни одного человека, который может разобраться во всей системе целиком. Ясно, что трудно спроектировать систему хорошо, если никто на самом деле не понимает её.

Это приводит нас ко второй упомянутой выше особенности: локализации ошибок. Например, ни один человек также не разбирается абсолютно во всех деталях того, как функционирует авианосец, да это и не нужно. Просто подсистемы авианосца хорошо изолированы друг от друга. Поэтому проблема с засорённым туалетом не может повлиять, например, на подсистему запуска ракет.

Операционные же системы не имеют столь сильной изоляции между компонентами. Современная операционная система содержит сотни тысяч процедур, скомпонованных вместе как единая двоичная программа, функционирующая в режиме ядра. Каждая из миллионов строк кода ядра может переписывать основные структуры данных, используемых независимыми компонентами ядра, выводя систему из строя способами, которые трудно обнаружить. Кроме того, если вирус или червь инфицирует одну процедуру ядра, то не существует способа удержать его от быстрого распространения на остальные процедуры и от захвата управления машиной в целом.

Возвращаясь к нашей корабельной аналогии: современные корабли имеют множество отсеков внутри корпуса; если один отсек даст течь, то затопится только он один, а не весь корабль. Современные операционные системы выглядят как корабли до изобретения водонепроницаемых переборок: любая течь может утопить корабль.

К счастью, ситуация не безнадёжна. Исследователи пытаются создать более надёжные операционные системы. В этой статье мы рассмотрим четыре различных подхода, используемых исследователями для того, чтобы сделать операционные системы будущего более надёжными и безопасными, начиная с наименее радикального и заканчивая наиболее радикальным решением.

БРОНИРОВАННЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

Наиболее консервативный подход, Nooks[4] (труднодоступные места), разработан для улучшения надёжности существующих операционных систем, таких, как Windows или Linux. Nooks сохраняет в работоспособном состоянии структуру монолитного ядра с сотнями тысяч процедур, скомпонованных вместе в единое адресное пространство в режиме ядра, однако этот подход концентрируется на том, чтобы сделать драйверы устройств – источник проблемы – менее опасными.

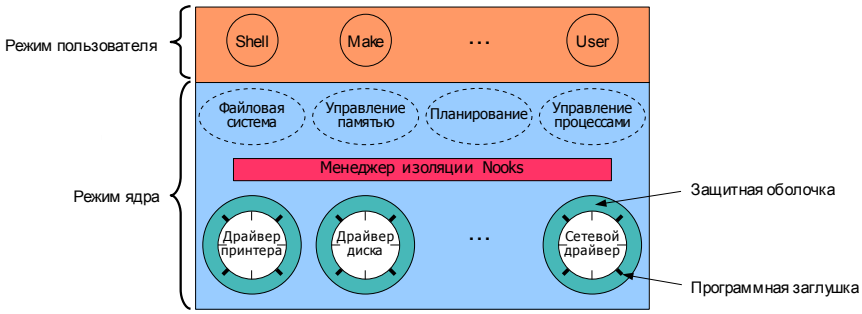

Рисунок 1. Модель Nooks. Каждый драйвер погружается в оболочку из защитного программного обеспечения, которое отслеживает все взаимодействия между драйвером и ядром.

В частности, как показано на Рисунке 1, Nooks защищает ядро от переполненных ошибками драйверов устройств, погружая каждый драйвер в оболочку из защитного программного обеспечения для формирования облегчённой защитной зоны; этот метод иногда называется «игра в песочнице». Оболочка вокруг каждого драйвера тщательно отслеживает все взаимодействия между драйвером и ядром. Этот метод также может быть использован для других расширений ядра, таких, как загружаемые файловые системы, однако для упрощения мы будем ссылаться только на драйверы.

Целями проекта Nooks являются:

- защита ядра от сбоев и отказов драйвера,
- автоматическое восстановление в случае сбоя драйвера, и
- делать всё это со столь минимально возможными изменениями по отношению к существующим драйверам и ядру.

Защита же ядра от драйверов-«злоумышленников» целью проекта не является. Первоначально метод был применён в Linux, но идеи в равной степени хорошо применимы и к другим традиционным ядрам.

Изоляция

Основным средством, используемым для защиты неисправных драйверов от мусора структур данных ядра, является виртуальная карта распределения страниц памяти. Когда драйвер работает, все страницы за его пределами изменяют статус на «read-only», таким образом формируя обособленную область легкой защиты для каждого драйвера. В этом случае драйвер может читать структуры данных ядра, которые ему нужны, однако любая попытка напрямую изменить структуру данных ядра приводит к нештатной ситуации центрального процессора, что отлавливается менеджером изоляции Nooks. Доступ к собственной памяти драйвера, в которой он сохраняет стеки, область динамически распределяемой памяти, собственные структуры данных и копии объектов ядра по-прежнему имеют статус «read-write».

Посредничество

Каждый класс драйверов экспортирует определённый набор функций, которые может запросить ядро. Например, звуковые драйверы могут отправить вызов для записи блока аудиоданных в аудиокарту, другой вызов может регулировать громкость и так далее. Когда драйвер загружен, заполняется массив указателей на функции драйвера; значит, ядро может найти каждую из них. Кроме того, драйвер импортирует набор функций ядра, например, для размещения буфера данных.

Nooks предусматривает защитные оболочки как для экспортируемых, так и для импортируемых функций. Когда ядро вызывает функцию драйвера или драйвер вызывает

функцию ядра, вызов на самом деле направляется в оболочку, которая проверяет параметры на достоверность и перенаправляет вызов. Несмотря на то, что программные заглушки оболочки – показанные на Рисунке 1 как отрезки, торчащие вовнутрь и вовне драйверов – генерируются автоматически из соответствующих функциональных прототипов, тела оболочек разработчики должны написать собственноручно. В целом, коллектив разработчиков Nooks написал 455 оболочек: 329 для экспортируемых функций ядра и 126 для функций, которые экспортируют драйверы устройств.

Когда драйвер пытается изменить объект ядра, его оболочка сначала копирует этот объект в защитную зону драйвера, то есть в его собственные страницы, имеющие статус «read-write». Затем драйвер модифицирует копию объекта. В случае успешного завершения вызова менеджер изоляции копирует изменённый объект ядра обратно в ядро. В этом случае отказ или повреждение драйвера во время вызова всегда оставляет объекты ядра неповреждёнными. Поскольку отслеживание импортируемых объектов специфично для различных объектов, коллектив разработчиков Nooks собственноручно написал код для отслеживания 43 классов объектов, которые используют драйверы Linux.

Восстановление

После возникновения сбоя запускается работающий в пользовательском режиме агент восстановления, который смотрит в конфигурационной базе данных, что нужно делать. Во многих случаях достаточно разблокировки каких-либо заблокированных ресурсов и перезапуска драйвера. Это объясняется тем, что наиболее общие алгоритмические ошибки обнаруживаются заранее, ещё во время тестирования, в ходе которого пропускаются в основном ошибки синхронизации и иные редко встречающиеся ошибки.

Этот метод может восстановить систему в целом, однако работающие приложения могут завершиться неудачно. В дополнительной работе[5] коллектив разработчиков Nooks добавил концепцию теневых драйверов для того, чтобы позволить приложениям продолжить работу и после сбоя драйвера.

Если вкратце, то во время нормального функционирования теневой драйвер регистрирует взаимодействие между каждым драйвером и ядром, если это будет необходимо для восстановления. После перезапуска драйвера теневой драйвер наполняет вновь запущенный драйвер информацией из журнала регистрации – например, повторяя системный вызов управления вводом-выводом (I/O control – IOCTL) для установки параметров, таких, например, как громкость звука. Ядро не знает о процессе перевода нового драйвера в состояние работы, в котором находился старый драйвер. Как только процесс восстановления завершается, драйвер начинает обрабатывать новые запросы.

Ограничения

Несмотря на то, что, согласно экспериментам, Nooks может ловить 99% фатальных ошибок драйвера и 55% некритических ошибок, до совершенства тут ещё далеко. Например, драйверы могут пытаться выполнять привилегированные инструкции, обработать которые они на самом не могут; они могут писать в ошибочные порты ввода-вывода; они, наконец, могут попадать в бесконечные циклы. Более того, поскольку коллектив разработчиков Nooks написал большое количество оболочек вручную, то и они также могут содержать ошибки. Наконец, драйверы не защищены от повторного доступа на запись всей памяти. Тем не менее, несмотря на всё вышесказанное, Nooks является потенциально полезным шагом в сторону улучшения надёжности существующих ядер.

ПАРАВИРТУАЛЬНЫЕ МАШИНЫ

Второй подход корнями восходит к концепции виртуальной машины, которая возвращает нас к концу 1960-х годов[6]. Вкратце, идея заключается в запуске специальной управляющей программы, называемой монитором виртуальной машины, напрямую на «голом железе», то есть на компьютере без операционной системы. Виртуальная машина создаёт множество копий реального компьютера. На каждой такой копии может функционировать любое программное обеспечение, которое может быть запущено на «голом» компьютере.

Эта методика обычно используется для того, чтобы позволить двум или большему количеству операционных систем (читай: Linux и Windows) запуститься одновременно на одном и том же компьютере, причём каждая ОС думает, что она владеет компьютером полностью и единолично. Использование виртуальных машин имеет заслуженную репутацию хорошего способа для локализации отказов – в конце концов, если ни одна из виртуальных машин даже не знает о существовании остальных, то и проблемы в одной из них не могут распространиться на остальные виртуальные машины.

Упомянутое здесь исследование призвано адаптировать эту концепцию для защиты внутри одной операционной системы, а не между различными операционными системами[7]. Более того, поскольку Pentium не является полностью виртуализируемым, было сделано допущение к идее запуска исходной (не модифицированной) операционной системы на виртуальной машине. Это допущение разрешает делать некоторые изменения в операционной системе с тем, чтобы удостовериться, что операционная система не делает ничего такого, что не может быть виртуализовано. Для того, чтобы отличать этот метод от истинной виртуализации, он называется *паравиртуализацией* (то есть необычной виртуализацией).

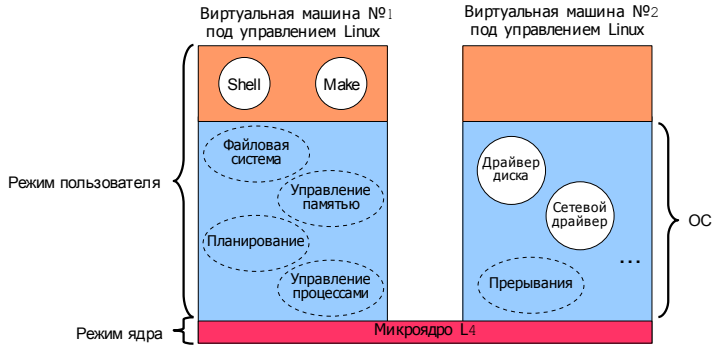

Рисунок 2. Виртуальные машины. Одна из виртуальных Linux машин выполняет прикладные программы в то время, как на другой (или других) машинах работают драйверы устройств.

Если конкретно, то в 1990-х исследовательская группа в университете Карлсруэ разработала микроядро L4[8]. Они смогли запустить слегка модифицированную версию Linux (L4 Linux) поверх L4, что было описано как разновидность виртуальной машины[9]. Позднее исследователи смогли вместо одной копии Linux запустить поверх L4 несколько копий. Как показано на Рисунке 2, интуиция привела их в дальнейшем к идее использовать одну из виртуальных машин для работы прикладных программ, а ещё одну или более виртуальных машин – для драйверов устройств.

Если поместить драйверы устройств в одну или более виртуальных машин, отделённых от основной виртуальной машины, на которой работает остальная часть операционной системы и прикладные программы, то в случае отказа какого-либо драйвера устройства «рухнет» только его виртуальная машина, но никак не основная. Дополнительное преимущество данного метода заключается в том, что нет никакой необходимости вносить изменения в драйверы устройств, поскольку они «видят» нормальное состояние ядра Linux. Конечно же, само ядро Linux должно было быть изменено в целях паравиртуализации, но это разовое изменение, и нет необходимости повторять его для

каждого драйвера устройства.

Так как драйверы устройств запущены в аппаратном пользовательском режиме (hardware's user mode), то главной проблемой является то, как они на самом деле осуществляют ввод-вывод и обрабатывают прерывания. Физический ввод-вывод обеспечивается добавлением около 3'000 строк кода к ядру Linux, поверх которого работают драйверы, для того, чтобы разрешить им использовать функции L4 по вводу-выводу вместо того, чтобы делать это самостоятельно. Дополнительные 5'000 строк кода обеспечивают взаимодействие между тремя изолированными драйверами – диска, сети и шины PCI – и виртуальной машиной, на которой выполняются прикладные программы.

В принципе, этот подход должен обеспечить большую надёжность, нежели одна операционная система, поскольку когда виртуальная машина содержит один или более аварийных драйверов, она может быть перезагружена и драйверы вернутся в первоначальное состояние. Однако не делается никаких попыток вернуть драйверы в предыдущее (предаварийное) состояние, как это делается в Nooks. Таким образом, если «рухнул» аудио драйвер, он будет восстановлен с уровнем громкости «по умолчанию», а не с уровнем, который был до сбоя.

Измерения быстродействия показали, что потери производительности от использования паравиртуализированных таким образом машин составляют от 3 до 8 процентов.

МУЛЬТИСЕРВЕРНЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

Первые два подхода основываются на латании существующих операционных систем. Следующие же два ориентированы на системы будущего.

Один из этих подходов нацелен непосредственно на корень проблемы: существование цельной операционной системы, запускающейся как единая гигантская двоичная программа в режиме ядра. И это вместо того, чтобы только крошечное микроядро запускалось в режиме ядра, а вся остальная часть операционной системы работала в режиме пользователя как набор полностью изолированных серверных процессов и драйверов.

Этой идее около 20 лет, однако она не была полностью исследована сразу же, поскольку имела несколько меньшую производительность, нежели монолитное ядро. В 1980-е производительность подсчитывалась «везде и всюду», а вот надёжность и безопасность ещё не были под жёстким контролем. Конечно, в своё время авиационные инженеры тоже не слишком сильно беспокоились по поводу расхода топлива или способности двери в кабине самолёта противостоять вооружённым атакам. Однако, времена меняются и человеческие представления о том, что является важным, меняются тоже.

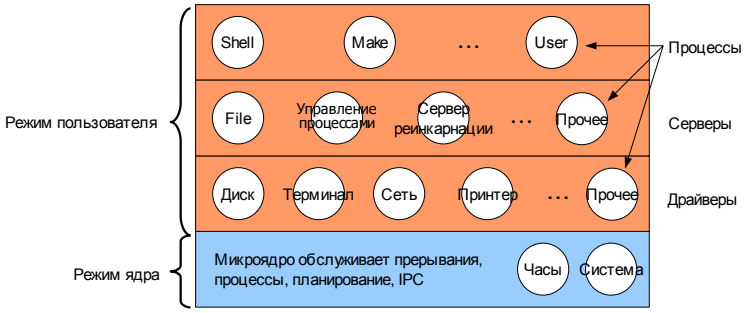

Рисунок 3. Архитектура Minix 3. Микроядро обрабатывает прерывания, обеспечивает основные механизмы управления процессами, осуществляет межпроцессное взаимодействие и выполняет планировку процессов.

Мультисерверная архитектура

Рассмотрение нового примера позволяет сделать идею мультисерверной операционной системы яснее. Как показано на Рисунке 3, в Minix 3 микроядро обрабатывает прерывания, обеспечивает основные механизмы управления процессами, осуществляет межпроцессное взаимодействие и выполняет планировку процессов. Оно также предлагает небольшой набор вызовов ядра для авторизации драйверов и серверов, таких, как чтение отмеченной области абсолютного адресного пространства пользователя или запись в санкционированные порты ввода-вывода. Драйвер часов также использует адресное пространство микроядра, однако он планируется как отдельный процесс. Никакие другие драйверы в режиме ядра не запускаются.

Поверх микроядра находится уровень драйверов устройств[10]. Каждое устройство ввода-вывода имеет свой собственный драйвер, который работает как отдельный процесс в своём собственном адресном пространстве, защищённом аппаратным блоком управления памятью. Уровень включает драйверные процессы для диска, терминала (клавиатуры и дисплея), Ethernet, принтера, аудио и т.п. Драйверы запускаются в режиме пользователя и не могут исполнять привилегированные команды или осуществлять чтение-запись из/в компьютерные порты ввода-вывода; они должны выдать запрос ядру для получения этих сервисов. Несмотря на относительно небольшие привносимые издержки производительности, эта схема существенно повышает надёжность.

Поверх уровня драйверов устройств находится уровень серверов. Файловый сервер является маленькой (4'500 строк исполняемого кода) программой, которая принимает запросы от пользовательских процессов на формирование POSIX-системных вызовов, связанных с файлами, таких, как read, write, lseek и stat, и выполняет их. На этом уровне также находится диспетчер процессов, который осуществляет управление процессами и памятью, и выполняет POSIX- и другие системные вызовы, такие, как fork, exec и brk.

Несколько необычным является сервер реинкарнации (reincarnation server), который является родительским процессом всех остальных серверов и всех драйверов. Если драйвер или сервер «падают», завершают работу или не отвечают на периодические запросы, сервер реинкарнации при необходимости останавливает (точнее сказать, «убивает») их, а затем заново запускает их из копии на диске или в оперативной памяти. В настоящее время таким способом могут быть перезапущены любые драйверы, а также только те серверы, которые не обслуживают большой объём внутрисистемных процессов.

Остальные серверы включают сетевой сервер, который содержит полный стек TCP/IP, информационное хранилище (the data store), простой сервер преобразования имён, который используют все остальные серверы, и информационный сервер, помогающий при отладке.

И наконец, над серверным уровнем располагаются пользовательские процессы. Единственная разница между этой и остальными Unix-системами заключается в том, что библиотечные процедуры для read, write и остальных системных вызовов выполняются через посылку сообщений серверам. Кроме этой разницы – скрытой в системных библиотеках – они являются нормальными пользовательскими процессами, которые могут использовать POSIX API.

Межпроцессное взаимодействие

В связи с тем, что межпроцессное взаимодействие (interprocess communication – IPC) позволяет всем процессам взаимодействовать друг с другом, оно является элементом чрезвычайной важности в мультисерверной операционной системе. Поскольку все серверы и драйверы в Minix 3 работают как физически изолированные процессы, они не могут вызывать функции друг друга непосредственно или совместно использовать какие-либо структуры данных. Вместо этого Minix 3 реализует IPC, посылая сообщения фиксированной длины по принципу рандеву: когда и отправитель, и получатель готовы,

система копирует сообщение непосредственно от отправителя к получателю. Кроме того, применяется и механизм асинхронного уведомления о событии. События, которые не могут быть отработаны, помечаются в таблице процессов как ожидающие.

Minix 3 изящно объединяет прерывания с системой передачи сообщений. Обработчики прерываний используют механизм уведомлений для сигнализации о завершении ввода-вывода. Этот механизм позволяет обработчику установить бит «отложенное прерывание» в битовой карте драйвера и затем продолжить работу без блокировки. Когда драйвер готов к получению прерывания, ядро превращает его в обычное сообщение.

Характеристики надёжности

Надёжность Minix 3 обеспечивается многими факторами. Прежде всего, только около 4'000 строк кода запускается в режиме ядра, так что оценивая с запасом количество ошибок в 6 штук на 1000 строк кода, общее число ошибок в ядре, вероятно, составляет около 24 штук – по сравнению с 15'000 для Linux и гораздо большим их количеством для Windows. Поскольку все драйверы устройств, кроме часов, являются пользовательскими процессами, никакой посторонний код никогда не запускается в режиме ядра. Маленький размер ядра также позволяет на практике целиком проверить его код как вручную, так и формальными методами.

Схема IPC в Minix 3 не требует очередей или буферизации сообщений, что исключает потребность управления буфером в ядре. Более того, поскольку IPC представляет собой мощную конструкцию, возможности каждого сервера или драйвера по межпроцессному взаимодействию сильно урезаны. Для каждого процесса доступные функции IPC, разрешённые адресаты и пользовательские уведомления о событии ограничены. Процессы пользователя, например, могут использовать только принцип рандеву и могут адресоваться только к POSIX-серверам.

Кроме того, все структуры данных ядра являются статическими. Все эти особенности сильно упрощают код и устраняют ошибки, связанные с переполнениями буфера, утечками информации из памяти, несвоевременными прерываниями, непроверенным кодом ядра и проч. Конечно же, перевод большей части операционной системы в режим пользователя не устранит неизбежные ошибки в драйверах и серверах, но сделает их гораздо менее действенными. Ошибка в ядре может разрушить критические структуры данных, записать какой-нибудь «мусор» на диск и так далее; ошибка в большинстве драйверов и серверов не может нанести таких больших повреждений, поскольку эти процессы очень строго разделены и очень сильно ограничены в части того, что они могут делать.

Драйверы и сервера в режиме пользователя не могут запускаться с привилегиями суперпользователя. Они не могут получить доступ к памяти за пределами собственного адресного пространства иначе, как через вызовы ядра (которые ядро проверяет на достоверность). Вдобавок, битовые карты и диапазоны внутри таблицы процесса ядра управляют набором разрешённых вызовов ядра, возможностями IPC и открывают порты ввода-вывода на препроцессной основе. Например, ядро может помешать драйверу принтера произвести запись в адресные пространства пользователя, обращаться в дисковые порты ввода-вывода или послать сообщение аудио драйверу. В классических монолитных системах любой драйвер может делать всё, что угодно.

Другое средство обеспечения надёжности – это использование отдельных областей памяти для данных и команд. Даже если ошибка или вирус приведёт к переполнению буфера драйвера или сервера и разместит чужой код в области данных, внедрённый код не сможет быть исполнен ни прямой передачей управления ему, ни в ходе процедуры выхода из прерывания и возврата к нему, поскольку ядро не может запустить на исполнение код, если он не находится в области команд процесса (имеющей, кстати, статус «read-only»).

Среди прочих специфических особенностей, помогающих улучшить надёжность, наиболее важной является способность к самовосстановлению. Если драйвер сохраняет что-либо через неверный указатель, попадая в бесконечный цикл или ведёт себя неправильно каким-либо другим способом, то сервер реинкарнации автоматически заменит его, часто даже незаметно для запущенных процессов.

До тех пор, пока перезапуск логически некорректного драйвера не устранит ошибку, коварные ошибки синхронизации (или аналогичные им) на самом деле доставляют множество проблем; перезапуск же драйвера часто восстанавливает систему. Кроме того, этот механизм обеспечивает восстановление после отказов в работе, вызванных атаками, такими, как «запрос смерти» («ping of death»), который может «обрушить» компьютер, посылая ему некорректно сформированный IP-пакет.

Соображения производительности

Десятилетиями исследователи критиковали мультисерверные архитектуры, основанные на микроядрах, из-за предполагаемых проблем с производительностью. Однако, различные исследования доказали, что модульные конструкции на самом деле могут обеспечить вполне конкурентную производительность. Несмотря на то, что Minix 3 не оптимизировался с точки зрения производительности, система достаточно быстра. Потеря производительности, к которой приводит перемещение драйверов из ядра в пространство пользователя, составляет менее 10 процентов; система, например, может собрать себя (включая ядро, стандартные драйверы и все серверы - 112 компиляций и 11 компоновок) на 2,2-гигагерцовом процессоре Athlon менее, чем за 6 секунд.

Тот факт, что мультисерверные архитектуры позволяют обеспечить сверхнадёжную Unix-среду ценой небольшой потери производительности, делает практическую реализацию этого подхода целесообразной. Minix 3 для Pentium доступен для свободной загрузки под лицензией Berkeley на сайте www.minix3.org. Порты на остальные архитектуры и на встраиваемые системы находятся на стадии разработки.

ЗАЩИТА СРЕДСТВАМИ ЯЗЫКА

Самый радикальный подход появился с совершенно неожиданной стороны – из Microsoft Research. В сущности, подход Microsoft отвергает концепцию операционной системы как единой программы, запущенной в режиме ядра, с некоторым набором пользовательских процессов, запущенных в режиме пользователя, и заменяет его на систему, написанную на новом типе языков, обеспечивающих типовую безопасность (type-safe language), которые не имеют всех этих связанных с C и C++ проблем с указателями и проч. Как и двум предыдущим, этому подходу тоже несколько десятков лет.

Этот подход использовался в машине Burroughs B5000. Единственным доступным для неё языком был Algol, при этом защита обеспечивалась не диспетчером памяти (MMU) – машина его не имела, а отказом компилятора Algol компилировать «опасный» код. Подход Microsoft Research только реанимирует эту идею применительно к 21 веку.

Обзор

Эта система, названная Singularity, написана почти полностью на Sing# - новом языке, обеспечивающем типовую безопасность. Язык основан на C#, но усилен примитивами передачи сообщений (message passing primitives), чья семантика определяется формальными, зафиксированными в исходном тексте, соглашениями (written contracts). Поскольку языковая безопасность накладывает очень сильные ограничения на систему и пользовательские процессы, то все процессы могут запускаться вместе в одном и том же виртуальном адресном пространстве. Это решение приводит к повышению как безопасности (поскольку компилятор не позволит одному процессу трогать данные другого процесса), так и производительности, поскольку оно устраняет системные

прерывания и контекстные переключатели.

Более того, модель Singularity достаточно гибка, поскольку каждый процесс является замкнутым объектом (closed entity) и, таким образом, может иметь свой собственный код, структуры данных, карту размещения в памяти, исполняемую систему (runtime system), библиотеки и «сборщик мусора» (garbage collector). Использование диспетчера памяти (MMU) допускается, но только для управления отображением (to map) страниц памяти, а не для установки отдельной области защиты для каждого процесса.

Ключевой замысел Singularity заключается в запрете динамических расширений процесса (dynamic process extensions). Помимо прочих особенностей, Singularity не допускает существование подгружаемых модулей, таких, как драйверы устройств и плагины браузера, поскольку они могут привнести непроверенный внешний код, способный повредить материнский процесс. Вместо этого такие расширения должны запускаться как отдельные процессы, полностью изолированные и взаимодействующие через стандартный механизм IPC.

Микроядро

Операционная система Singularity состоит из процесса микроядра и набора пользовательских процессов, обычно запускаемых в общем виртуальном адресном пространстве. Микроядро управляет доступом к оборудованию, распределяет и освобождает память, создаёт, уничтожает и планирует потоки (threads), управляет синхронизацией потоков с помощью семафоров (mutexes), управляет межпроцессной синхронизацией через каналы и «заведует» вводом-выводом. Каждый драйвер устройства работает как отдельный процесс.

Хотя большая часть микроядра написана на Sing#, небольшая его часть написана на C#, C++ или ассемблере и должна быть надёжной, поскольку не может быть верифицирована формальными автоматическими способами. Выверенный код включает в себя уровень аппаратных абстракций (hardware abstraction layer) и «сборщик мусора» (garbage collector). Уровень аппаратных абстракций прячет от системы низкоуровневое оборудование, скрывая такие понятия, как порты ввода-вывода, линии запроса прерывания, каналы прямого доступа к памяти и таймеры, для того, чтобы представить машинно-независимые абстракции остальной части операционной системы.

Межпроцессное взаимодействие

Пользовательские процессы получают обслуживание от системы (system services) в ответ на отправку микроядру строго формализованных (strongly typed) сообщений по двунаправленным радиальным (типа «точка-точка») каналам. Фактически всё взаимодействие между процессами (process-to-process communication) использует эти каналы. В отличие от других систем передачи сообщений, имеющих функции SEND и RECEIVE в некоей библиотеке, Sing# поддерживает каналы полностью на уровне языка, включая формальные спецификации типов данных и протоколов (typing and protocol specifications).

Для пояснения рассмотрим такую спецификацию канала:

```
contract C1 {
in message Request(int x) requires x > 0;
out message Reply(int y);
out message Error();

state Start:
Request? -> Pending;
state Pending: one {
```

```
Reply! -> Start;
Error! -> Stopped;
}
state Stopped: ;
}
```

Это соглашение (contract) объявляет, что канал принимает три сообщения, «Request», «Reply» и «Error»; первое - с положительным целым в качестве параметра, второе – с любым целым в качестве параметра, а третье – без параметров. При использовании в качестве канала связи с сервером сообщения «Request»идут от клиента к серверу, а остальные два сообщения идут в обратном направлении. Механизм задания состояний (state machine) точно определяет протокол для канала.

В состоянии «Start»клиент посылает сообщение «Request», переводя канал в состояние «Pending»(«ждущий»). Сервер может ответить сообщениями «Reply»или «Error». Сообщение «Reply»переводит канал в первоначальное состояние «Start», в котором взаимодействие может быть продолжено. Сообщение «Error»переводит канал в состояние «Stopped»(«остановленный»), завершая взаимодействие по каналу.

Область динамически распределяемой памяти (heap - «куча»)

Если все данные, такие, как блоки данных, считанные с диска, будут передаваться по каналам, то система будет чрезвычайно медленной. Поэтому из основополагающего правила (процесс является замкнутым объектом, поэтому любые данные процесса являются только его собственными внутренними данными) было сделано исключение. Singularity поддерживает разделяемую область динамически распределяемой памяти для объектов (shared object heap), но в каждый момент времени каждый объект в «куче» принадлежит только одному процессу. Однако право собственности на объект может передаваться по каналу.

В качестве примера того, как работает «куча», рассмотрим ввод-вывод. Как только драйвер диска прочитает блок, он помещает этот блок в «кучу». Затем система передаёт указатель на блок пользователю, запросившему эти данные, сохраняя принцип одного пользователя (single-owner principle), но позволяя данным перемещаться с диска к пользователю без лишнего копирования.

Файловая система

Singularity поддерживает единое иерархическое пространство имён (single hierarchical name space) для всех сервисов. Сервер преобразования root (root name server) обслуживает вершину дерева, а остальные серверы преобразования имён могут быть монтированы к его узлам. В частности, файловая система, которая является просто процессом, монтируется на /fs, так, что такое имя, как fs/users/linda/foo может оказаться файлом пользователя. Файлы реализованы в виде сбалансированных деревьев (B-trees) с номерами блоков в качестве ключей. Когда пользовательский процесс запрашивает файл, файловая система даёт команду дисковому драйверу положить запрошенные блоки в «кучу». Право собственности затем передаётся, как описано выше.

Верификация

Каждый компонент системы имеет метаданные, описывающие его зависимости, выходные параметры (exports), входные параметры (resources) и режим работы (behavior). Эти метаданные используются для формальной автоматической верификации. Образ системы включает в себя микроядро, драйверы и приложения, необходимые для работы системы, вместе с их метаданными. Внешние верификаторы могут выполнять множество проверок на образе (таких, как подтверждение того, что драйверы не имеют конфликтов на уровне использования ресурсов) прежде, чем система запустит его на исполнение.

Верификация является трёхшаговым процессом:

- Компилятор проверяет типовую безопасность, право собственности на объект, протоколы каналов и проч.
- Компилятор генерирует Microsoft Intermediate Language (MSIL) – компактный байтовый код в стиле Java Virtual Machine, который верификатор может проверить.
- MSIL компилируется в код x86 вспомогательным компилятором (back-end compiler), который может добавлять в код динамические проверки (основной компилятор этого не делает).

Целью избыточной проверки является обнаружение ошибок в самих верифицирующих инструментах.

Каждая из четырёх различных попыток улучшить надёжность операционной системы базируется на предотвращении полного отказа системы из-за множества ошибок в драйверах устройств.

В подходе Nooks каждый драйвер персонально «одевается» в программную защитную «курточку» для того, чтобы тщательно контролировать его взаимодействие с оставшейся частью операционной системы, но при этом все драйверы остаются в ядре. Подход в виде паравиртуальной машины принимает более «продвинутые» меры и перемещает драйверы в одну или более виртуальных машин, отличных от основной, ещё больше ограничивая «вредоносные способности» драйверов. Оба эти подхода предназначены для улучшения надёжности существующих операционных систем.

В отличие от них, два других подхода предлагают заменить существующие операционные системы на более надёжные и безопасные. Мультисерверный подход запускает каждый драйвер и компоненту операционной системы в отдельном пользовательском процессе и разрешает им взаимодействовать друг с другом только через IPC-механизм микроядра. И, наконец, Singularity, как наиболее радикальный подход, использует язык, обеспечивающий типовую безопасность (type-safe language), единое адресное пространство и формальные соглашения (contracts), зафиксированные в исходном тексте, для того, чтобы тщательно ограничить возможности каждого модуля.

Три из четырёх исследовательских проектов – паравиртуализация на базе L4, Minix 3 и Singularity – используют микроядро. Пока неясно, какой из этих подходов (если вообще какой-нибудь из них) получит в конце концов широкое распространение. Тем не менее, интересно отметить, что микроядра – длительное время отвергаемые из-за их меньшей производительности по сравнению с монолитными ядрами – вновь могут вернуться благодаря их потенциально более высокой надёжности, которую многие люди сегодня рассматривают как более важную характеристику, чем производительность. Колесо реинкарнации повернулось.

Благодарности

Мы благодарим Брайана Бершада (Brian Bershad), Галена Ханта (Galen Hunt) и Майкла Свифта (Michael Swift) за их комментарии и предложения. Эта работа была частично поддержана Netherlands Organization for Scientific Research в рамках гранта 612-060-420.

Ссылки

1. V.R. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Comm. ACM*, Jan. 1984, pp. 42-52.
2. T.J. Ostrand and E.J. Weyuker, The Distribution of Faults in a Large Industrial Software System, *Proc. Int'l Symp. Software Testing and Analysis*, ACM Press, 2002, pp. 55-64.
3. A. Chou et al., "An Empirical Study of Operating System Errors," *Proc. 18th ACM Symp. Operating*

System Principles, ACM Press, 2001, pp. 73-88.
4. M. Swift, B. Bershad, and H. Levy, " Improving the Reliability of Commodity Operating Systems," *ACM Trans. Computer Systems*, vol. 23, 2005, pp. 77-110.
5. M. Swift et al., "Recovering Device Drivers," *Proc. 6th Symp. Operating System Design and Implementation*, ACM Press, 2003, pp. 1-16.
6. R.P. Goldberg, "Architecture of Virtual Machines," *Proc. Workshop Virtual Computer Systems*, ACM Press, 1973, pp. 74-112.
7. J. LeVasseur et al., "Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines," *Proc. 6th Symp. Operating System Design and Implementation*, 2004, pp. 17-30.
8. J. Liedtke, "On Microkernel Construction," *Proc. 15th ACM Symp. Operating System Principles*, ACM Press, 1995, pp. 237-250.
9. H. Hartig et al., "The Performance of Microkernel-Based Systems," *Proc. 16th ACM Symp. Operating System Principles*, ACM Press, 1997, pp. 66-77.
10. J.N. Herder et al., "Modular System Programming in MINIX 3," Usenix; www.usenix.org/publications/login/2006-04/openpdfs/herder.pdf.

Эндрю С. Таненбаум (Andrew S. Tanenbaum) является профессором в области компьютерных наук (computer science) университета Врийе, Амстердам. Его научные интересы связаны с операционными системами и компьютерной безопасностью. Таненбаум получил степень бакалавра в Массачусетском Технологическом Институте и степень доктора философии в Калифорнийском Университете в Беркли. Он является членом Института инженеров по электротехнике и электронике (IEEE) и членом Ассоциации по вычислительной технике (ACM). Связаться с ним можно по адресу ast@cs.vu.nl.

Йоррит Н. Эрдер (Jorrit N. Herder) в настоящее время получает степень доктора философии на отделении компьютерных систем Департамента компьютерных наук университета Врийе, Амстердам. Его научные интересы связаны с разработкой и применением безопасных и надёжных операционных систем. Эрдер получил степень магистра естественных наук в университета Врийе. Связаться с ним можно по адресу jnherder@cs.vu.nl.

Герберт Бос (Herbert Bos) является доцентом отделения компьютерных систем Департамента компьютерных наук университета Врийе, Амстердам. Его научные интересы включают современные средства организации сетей (advanced networking technology), операционные системы и компьютерная безопасность. Бос получил степень доктора философии в Кембриджском университете. Связаться с ним можно по адресу bos@cs.vu.nl.